

CS 5220: Homework 2

Group 19: Robert Carson (rac428), Robert Chiodi (rmc298), Sam Tung (sat83)

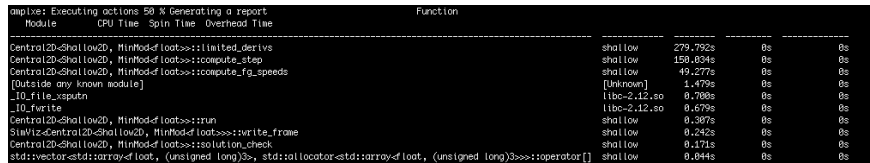
October 17, 2015

1 Initial Profiling

In order to perform initial profiling of the code before any improvements are made, Intel's VTUNE was used via the terminal command line on Totient. In an attempt to get the most accurate results (taken with a large sample size), we decided to gather data while running the large wave simulation, invoked with the command `make big`. Information collected from VTUNE's 'advanced-hotspots' option is used in our analysis.

1.1 Whole Program - Advanced Hotspots

First, hotspots in the entire program were examined in order to determine where our efforts should be directed. The time taken in the top 10 most time consuming functions can be seen below in Figure 1. Of these, it is clear that most of our optimization efforts should be directed to the functions `limited_derivs`, `compute_step`, and `compute_fg_speeds`. These functions were then examined individually, once again using VTUNE on our advanced-hotspots collection.



Module	CPU Time	Spin Time	Overhead Time	Function
Central2D-Shallow2D, MinModOfloats::limited_derivs	shallow	279.792s	0s	0s
Central2D-Shallow2D, MinModOfloats::compute_step	shallow	150.084s	0s	0s
Central2D-Shallow2D, MinModOfloats::compute_fg_speeds	shallow	49.277s	0s	0s
[Outside any known module]	[Unknown]	1.479s	0s	0s
_IO_file_xsputn	libc-2.12.so	0.700s	0s	0s
_IO_fwrite	libc-2.12.so	0.679s	0s	0s
Central2D-Shallow2D, MinModOfloats::run	shallow	0.597s	0s	0s
SimViz-Central2D-Shallow2D, MinModOfloats::write_frame	shallow	0.242s	0s	0s
Central2D-Shallow2D, MinModOfloats::solution_check	shallow	0.171s	0s	0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]	shallow	0.044s	0s	0s

Figure 1: Top 10 most time consuming functions in the wave simulation. Generated using Intel's VTUNE on Totient.

1.2 `limited_derivs` - Advanced Hotspots

The function `limited_deriv` is used to calculate the fluxes into and out of each cell in order to advance to the next time step. This involves a three point computational stencil in each direction and loops through the entire domain interior (the whole domain except for those where boundary conditions are applied). Each point requires `du.size() × 9` floating point operations as well as `du.size() × 2` calls to the intrinsic function `min`. Sadly, the hotspot

analysis on the `limited_deriv` function, shown in Figure 2, does not give any hints on possible optimizations or bottle necks.

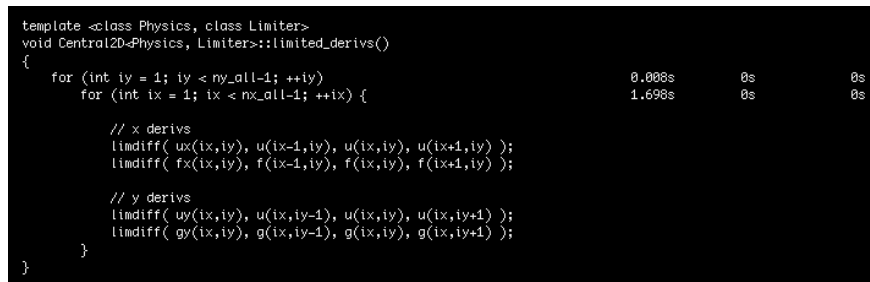


Figure 2: Time taken to perform each loop present in `limited_derivs`.

1.3 `compute_step` - Advanced Hotspots

The purpose of `compute_step` is to update the wave equation to the next time step using a predictor-corrector method. First, the fluxes are calculated in the prediction. Next, the corrector step uses the predicted fluxes, the differences in velocities, and the current velocities to advance to the next time state. Luckily, VTUNE’s report is more helpful than in the previous case, and provides extensive timings for this function, shown in Figure 3. The calculation in the corrector step can be seen to be the most expensive cost of the function. It is important to note, however, that the predictor step and copying of the solution to the `u` array sum to half of the function’s cost.

1.4 `compute_fg_speeds` - Advanced Hotspots

The function of `compute_fg_speeds` has two primary responsibilities: to update the cell centered fluxes, `f` and `g`, and to calculate the maximum speed in the domain, allowing dynamic adjustment of the time step in order to satisfy the CFL condition and ensure numerical stability. The timing data for this function can be seen in Figure 4. While the most time consuming portion of the code is most likely the calculation of the fluxes and wave speed (both of which are in the `Shallow2d` structure), the calls to the intrinsic function `max` also represent a non-trivial amount of time.

2 Parallelization

While we are aware that the optimal programming of this code will not wholly be due to parallelization by OpenMP, it is most certainly necessary. Furthermore, future tuning and optimization of the code may change depending on whether the code is run serially or parallel. For these reason, we first decided to parallelize the code and compare it’s performance against the initial, serial code.

template <class Physics, class Limiter>			
void Central2D<Physics, Limiter>::compute_step(int io, real dt)			
{			
real dtcdx2 = 0.5 * dt / dx;			
real dtcdy2 = 0.5 * dt / dy;			
// Predictor (flux values of f and g at half step)			
for (int iy = 1; iy < ny_all-1; ++iy)	0.003s	0s	0s
for (int ix = 1; ix < nx_all-1; ++ix) {	0.357s	0s	0s
vec uh = u(ix,iy);	1.909s	0s	0s
for (int m = 0; m < uh.size(); ++m) {			
uh[m] -= dtcdx2 * fx(ix,iy)[m];			
uh[m] -= dtcdy2 * gy(ix,iy)[m];	7.334s	0s	0s
}			
Physics::flux(f(ix,iy), g(ix,iy), uh);			
}			
// Corrector (finish the step)			
for (int iy = nghost-io; iy < ny+nghost-io; ++iy)	0.010s	0s	0s
for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {	1.750s	0s	0s
for (int m = 0; m < v(ix,iy).size(); ++m) {			
v(ix,iy)[m] =	26.781s	0s	0s
0.2500 * (u(ix, iy)[m] + u(ix+1,iy) [m] +	1.553s	0s	0s
u(ix,iy+1)[m] + u(ix+1,iy+1)[m]) -	6.718s	0s	0s
0.0625 * (ux(ix+1,iy) [m] - ux(ix,iy) [m] +			
ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +			
uy(ix, iy+1)[m] - uy(ix, iy)[m] +			
uy(ix+1,iy+1)[m] - uy(ix+1,iy)[m]) -	1.501s	0s	0s
dtcdx2 * (f(ix+1,iy) [m] - f(ix,iy) [m] +	0.228s	0s	0s
f(ix+1,iy+1)[m] - f(ix,iy+1)[m]) -	7.972s	0s	0s
dtcdy2 * (g(ix, iy+1)[m] - g(ix, iy)[m] +	1.840s	0s	0s
g(ix+1,iy+1)[m] - g(ix+1,iy)[m]);	8.207s	0s	0s
}			
}			
// Copy from v storage back to main grid			
for (int j = nghost; j < ny+nghost; ++j){	0.003s	0s	0s
for (int i = nghost; i < nx+nghost; ++i){	1.624s	0s	0s
u(i,j) = v(i-io,j-io);	10.566s	0s	0s
}			
}			
}	0.001s	0s	0s

Figure 3: Time taken to perform each loop present in compute_step.

2.1 Naive Implementation of OpenMP

First, a naive implementation of OpenMP using pragmas was used. This involved placing `#pragma omp parallel for` in front of the start of each for loop in the functions `init`, `apply_periodic`, `compute_fg_speeds`, `limited_derivs`, and `compute_step`. This proved to improve performance significantly when run on one node of Totient (24 threads), as seen in Figure ??.

2.2 OpenMP for with **collapse**

“OpenMP parallel for” also has the `collapse` option, which essentially informs OpenMP how many nested loops are present. By knowing this, OpenMP is able to section both loops to run on multiple processors, creating blocks for each processor to work on. Results when using `collapse` can be seen in Figure ?. It was found that using the `collapse` option actually led to worse performance. We believe this is due to the creation of blocks, which will limit the amount of memory accesses each core has that is of unit stride. When only sectioning the for loops based on the vertical (y) direction, each processor gets a $n \times nx$ block of the array, where n is some number of rows (vertical lines of computational cells) in the domain. This increases memory access locality, which in turn increases cache hits.

{			
using namespace std;			
real cx = 1.0e-15;			
real cy = 1.0e-15;			
for (int iy = 0; iy < ny_all; ++iy)	0.001s	0s	0s
for (int ix = 0; ix < nx_all; ++ix) {	3.248s	0s	0s
real cell_cx, cell_cy;			
Physics::flux(f(ix,iy), g(ix,iy), u(ix,iy));			
Physics::wave_speed(cell_cx, cell_cy, u(ix,iy));			
cx = max(cx, cell_cx);	4.299s	0s	0s
cy = max(cy, cell_cy);	1.666s	0s	0s
}			
cx_ = cx;			
cy_ = cy;			
}			

Figure 4: Time taken to perform each loop present in `compute_fg_speeds`.

2.3 OpenMP thread creation limited:

The constant creation of threads creates a large overhead cost that negatively effects the performance of the program. Therefore, the fewer times that a pool of threads needs to be created the better. In the previous examples, the thread pools were created each time a parallel for loop was called. In order to reduce this costly operation, a pool of threads was only created within the main run function as shown in the below code snippet.

```
for (int io = 0; io < 2; ++io) {
    real cx, cy;
    #pragma omp parallel
    {
        apply_periodic();
        compute_fg_speeds(cx, cy);
        limited_derivs();
        #pragma omp single
        {
            if (io == 0) {
                dt = cfl / std::max(cx/dx, cy/dy);
                if (t + 2*dt >= tfinal) {
                    dt = (tfinal-t)/2;
                    done = true;
                }
            }
        }
        compute_step(io, dt);
        #pragma omp single
        t += dt;
    }
}
```

Sections of the code that only required one thread to execute took advantage of the `omp single` pragma command which tells only one thread to run that section of code while the rest wait for it to finish. Inside each of the computation calls the regular `omp for` pragma command is used to parallelize the various for loops being run. The performance of this

operation for the top five costliest function calls when run on the big simulation can be found in Figure 5.

Function	Module	CPU Time
[Events Lost On Trace Overflow]	Events Lost On Trace Overflow	176.770s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	100.152s
<u>omp_barrier</u>	libiomp5.so	98.943s
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	76.487s
<u>omp_hyper_barrier_release</u>	libiomp5.so	30.561s
[Unknown stack frame(s)]	[Unknown]	22.861s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	20.214s

Figure 5: Timing of five costliest functions using fewer thread creation events. Generated using Intel’s VTUNE on Totient.

It should also be noted that these results do include the effects of attempting to force the compiler to vectorize several loops by replacing the min and max calls with an equivalent if statement min and max code. Also from Figure 5, it can be seen that the program still spends several a large amount of time in block operations.

!Not sure how it compares yet to the previous case need to look at those results, but it should be better

2.4 OpenMP nowait implementation:

When using the `omp for` pragma command, an implicit barrier is placed at the end of the loops. Due to the nature of this program, it is possible in several places to be able to assume that a barrier is not needed, and therefore the `nowait` pragma command can be used. The `nowait` command could be safely used in the `compute_fg_speeds` and `limited_derivs` functions. Then it could be partially used in the `compute_step` function once the predictor step was calculated. If the `nowait` command was used for the predictor loop it is possible for a segmentation fault to occur. If a thread was able to go onto the corrector step while other threads were still in the predictor step leading to errors. The performance of this implementation for the top five costliest function calls when run on the big simulation can be found in Figure 6

Function	Module	CPU Time
[Events Lost On Trace Overflow]	Events Lost On Trace Overflow	173.308s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	109.107s
<u>omp_barrier</u>	libiomp5.so	73.633s
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	63.533s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	40.993s

Figure 6: Timing of five costliest functions using `nowait` pragma command. Generated using Intel’s VTUNE on Totient.

It can be seen by comparing Figure 5 and Figure 6 that the overall time spent in the barrier function call has gone down drastically by including the `nowait` call.

2.5 Thread Affinity:

OpenMP 4.0 offers the ability to describe the ordering of the threads among the CPUs/MICs. Threads can be placed near each other so on the same CPU using the OpenMP pragma

command `proc_bind(close)`. The benefit of doing this is that amount of time for synchronization between threads should decrease. The negative downside of this is that the available cache and bandwidth per thread will decrease. Another option available distributes the threads more evenly throughout the available CPUs/MIC, and it can be turned on using the OpenMP pragma command `proc_bind(spread)`. It will lead to the opposite effects of the `close` command. The thread affinity was implemented on the code base that used the `nowait` pragma command. The performance of this implementation for the top five costliest function calls when run on the big simulation can be found in Figure 7 and Figure 8 for the `proc_bind(close)` pragma command. Due to system traffic, timing can vary across multiple runs. For this reason, it was decided to show two figures so some measure of variance could be judged.

Function	Module	CPU Time
[Events Lost On Trace Overflow]	Events Lost On Trace Overflow	166.097s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	106.580s
<u>km3c_barrier</u>	libomp5.so	81.897s
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	62.559s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	39.688s

Figure 7: Timing of five costliest functions using `proc_bind(close)` pragma command for run 1. Generated using Intel's VTUNE on Totient.

Function	Module	CPU Time
[Events Lost On Trace Overflow]	Events Lost On Trace Overflow	155.280s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	112.596s
<u>km3c_barrier</u>	libomp5.so	85.581s
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	65.491s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	43.297s

Figure 8: Timing of five costliest functions using `proc_bind(close)` pragma command for run 2. Generated using Intel's VTUNE on Totient.

Then results for the `proc_bind(spread)` command can be found in Figure 9.

Function	Module	CPU Time
[Events Lost On Trace Overflow]	Events Lost On Trace Overflow	119.259s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	104.188s
<u>km3c_barrier</u>	libomp5.so	100.178s
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	62.695s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	38.095s

Figure 9: Timing of five costliest functions using `proc_bind(spread)` pragma command. Generated using Intel's VTUNE on Totient.

Overall, the thread affinity options have shown to have led to a decreased performance. Therefore, they will not be used in the production code.

3 Analysis and Future Plans

The current implementations, while offering a sizable speed up over the strictly serial version, still leads much to be desired. Since it currently depends on the `parallel for` command to run with OpenMP automating certain commands, barriers are automatically placed into

the code that limits how each processor accesses the memory of the variables. Further improvements can be made to the code by eliminating these blocking structures where allowed and only syncing the data when the need arises in order to reduce the amount of overhead costs that occur from this type of synchronization. One such place where this could be beneficial is in the `compute_step` function. Specific structures could be used in order to ensure the predictor step is done for edge cases before the corrector begins without requiring a barrier. To do this, a more rigorous implementation of OpenMP will need to be done, which mimics more of a distributed memory parallel code. Blocks will be made of the domain, where each subdomain has ghost cells and is unaware of what happens outside of itself (except through the ghost cells). This gives more control and limits necessary communication.

Once the code has been parallelized with subdomains using OpenMP, use of the Xeon Phi chips will be implemented in the code. Thus far, we have had trouble using the Xeon Phi chips via the `#pragma offload target (mic:0)` command. We are not entirely sure of the issue, but believe it has to do with the use of templates in the C++ version of the code.

Once the code is parallelized into subdomains and effectively offloaded to the Xeon Phi chip, we will attempt to vectorize the loops. Preliminarily, we have had trouble vectorizing the loops within the C++ memory structure currently used, and have not managed to use the `restrict` keyword. Without this, the compiler is easily confused about dependencies inside the loop, as it cannot be sure where separate pointers are pointing to. It may be necessary to switch to the C version of the code that was written by Professor Bindel in order to tune the code for better vectorization.

3.1 Strong and Weak Scaling

The current version of the code can be analyzed for its strong and weak scaling capabilities. The strong scaling will be tested using the “big” wave test case with 1000 points in each direction and 100 time steps between frames. The results can be seen in Table 2. This is tested on the main Xeon chips. With the highest number of threads tested being 16, inter-node communication costs were avoided. VTUNE analysis was also not collected in order to remove any additional time the collection process takes. In Table 1, strong scaling is defined by Eq. 1 and normalized strong scaling is defined by Eq. 2.

$$\text{Strong Scaling} = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \quad (1)$$

$$\text{Normalized Strong Scaling} = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \frac{1}{p} \quad (2)$$

where t_{serial} and t_{parallel} are the times taken when running in serial or parallel, respectively, and p is the number of processors. The normalized strong scaling is essentially what percentage of perfect, linear scaling is actually achieved.

For weak scaling, three simulations were performed, scaling the number of processors equally with the number of cells. The results can be seen in Table ???. Here, weak scaling is defined by Eq. 3. It was decided to double the side length for each simulation in order to

Threads	Total Time (seconds)	Strong Scaling	Normalized Strong Scaling
1	539	1.0	100%
2	288	1.87	93.5%
4	218	2.47	61.7%
8	198	2.72	34.0%
16	188	2.87	17.9%

Table 1: Strong scaling for wave simulation with 1000 mesh points in each direction and 100 timesteps between frames.

perform the weak scaling test, meaning the number of processors needed to be quadrupled for each run.

$$\text{Weak Scaling} = \frac{t_{\text{serial}}(n(p))}{t_{\text{parallel}}(n(p), p)} \quad (3)$$

Threads	Cells Per Side	Total Time (seconds)	Weak Scaling
1	200	4.0	1.00
4	400	8.3	0.485
16	800	73.5	0.0544

Table 2: Strong scaling for wave simulation with 1000 mesh points in each direction and 100 timesteps between frames.

We believe the weak scaling performance is very poor due to our implementation of the OpenMP. Weak scaling usually tests the importance of communication on distributed memory systems, since each processor should be performing the same amount of algorithmic work as it is in the serial case. The main difference is the size of data being communicated is largely increased with the scaling in problem size. Since this is a shared memory code, we believe this indicates poor memory handling, leading to many cache misses. This is especially evident in the largest case, where we believe the problem size exceeded the cache size, causes very poor memory use. This could be fixed by a blocking scheme, where the domain is decomposed into subdomains, in order to increase memory locality, reducing the number of cache misses substantially. Another possible reason for the poor performance of the 16 thread case is an increase in time for each communication, due to the threads existing on two separate chips.

The scaling can be represented theoretically by creating an equation for the parallel time taken in the code. This would be primarily made up of two parts: the time to perform the floating point operations, and the portion spent communicating and synchronizing. In this code, since it uses shared memory, the only communications required involves OpenMP barriers and informing each processor of their section of the “for loops”. This theoretical relation can be seen in Eq. ??.

! Need to add this theoretical equation and explain

References